4

.NET Languages

Understanding a CLR-based language starts with understanding the CLR

The Common Language Runtime is explicitly designed to support multiple languages. In general, though, languages built on the CLR tend to have a good deal in common. By defining a large set of core semantics, the CLR also defines a large part of a typical programming language built using it. For example, a substantial chunk of learning any CLR-based language is seeing how the standard types defined by the CLR are mapped into that language. You must also learn the language's syntax, of course, including the control structures the language provides. Yet once you know what the CLR offers, you're a long way down the path to learning any language built on top of it.

This chapter describes C# and Visual Basic.NET, the most important CLR-based languages. It also takes a brief look at the Managed Extensions for C++ that allow C++ developers to write CLR-based code. The goal is not to provide exhaustive coverage of every language feature—that would require three more books—but rather to give you a sense of how these languages look and how they express the core functionality provided by the CLR.

What About Java for the .NET Framework?

In the fall of 2001, Microsoft announced Visual J#.NET, an implementation of the Java language built on the CLR. Despite this, I doubt that Java will ever be a viable choice for the .NET Framework. The reason is that even if a Java aficionado chooses to use a CLR-based Java compiler, such as Visual J#.NET, she's unlikely to be truly happy. Java implies a group of libraries and interfaces such as Swing and Enterprise JavaBeans. The .NET Framework provides its own equivalent technologies, so most of these won't be available. As a result, a developer using the Java language on the .NET Framework won't feel like she's working in a true Java environment because the familiar libraries won't be there.

The real target market for Visual J#.NET is Microsoft's existing Visual J++ customers. By providing a migration path to the .NET Framework, Microsoft is helping them move away from a dying product to one with a long future ahead of it. People who believe that Microsoft is truly interested in creating a first-class environment for building new Java applications might also wish to examine their beliefs about Santa Claus.

The battle lines are clear: It's .NET versus the Java world. This is unquestionably a good thing. Those who think everyone should implement Java forget both the dangers of monopoly and the sloth that comes with having no competition. Having two powerful technology camps, each with a strong position, is the ideal world. Each has innovations that the other can learn from, and each provides examples of things to avoid. In the end, the competition benefits everyone.

C#

C# is an object-oriented language with a C-like syntax

As its name suggests, C# is a member of the C family of programming languages. Unlike C, C# is explicitly object-oriented. Unlike C++, however, which is the most widely used object-oriented language in this family, C# isn't fiendishly complicated. Instead, C# was designed to be easily approachable by anyone with a background in C++ or Java.

Microsoft provides the dominant C# compilers but not the only ones

The most popular tool today for creating C# code is Microsoft's Visual Studio.NET. It's not the only choice, however. Microsoft also provides a command-line compiler with the .NET Framework called csc.exe, and the open source world has also created a C# compiler. Visual Studio.NET provides a rich environment for building CLR-based applications in C#, however, so it's hard to imagine that other alternatives will attract a large share of developers.

Standardizing C# and the CLR

Microsoft has submitted C# and a subset of the CLR called the Common Language Infrastructure (CLI) to the international standards body ECMA, where they are on track to become ECMA standards. Along with C#, the things that have been submitted for standardization include the syntax and semantics for metadata, MSIL (rechristened the Common Intermediate Language, or CIL), and parts of the .NET Framework class library. For more details on exactly what has been submitted and its current status, see http://msdn.microsoft.com/net/ecma.

Sun came close to doing something similar with its Java technology but backed away at the last minute. Will Microsoft's efforts be more successful? Sun resisted this step in large part because they were unwilling to give up control of Java. Control of Java is a valuable thing, and since Sun is a for-profit company, its reluctance to relinquish this control makes perfect sense. Microsoft is also a for-profit company. Will they really wait until ECMA has approved, say, an enhancement to C# before including it in their next release? And if they do, is this a good thing? Standards bodies aren't known for their speed.

I'd be surprised if Microsoft lets ECMA control the rate at which innovations appear in future releases of .NET technologies. Still, making C# and the CLI standards does give others a way to build them. Somewhat surprisingly, given its traditional antipathy toward Microsoft, the open source world has spawned various efforts to build parts of .NET. The most visible of these is the Mono project. (*Mono* means "monkey" in Spanish, which may be an oblique commentary on the Mono team's view of Microsoft.) Mono's ambitious goal is to implement at

least a large part of what Microsoft has given to ECMA, including a C# compiler and the CLI and perhaps more. Mono's creators say that they were attracted to the CLR for technical reasons, which must please Microsoft. In fact, from Mono's perspective, the CLI is the specification of a system while .NET's CLR is just the Microsoft implementation of this specification. Mono is certainly an interesting undertaking; to learn more about it, see http://www.go-mono.com.

Microsoft itself, together with Corel, has announced plans to make an implementation of C# and the CLI available for BSD UNIX. As discussed in Chapter 1, Microsoft faces substantial credibility problems in porting the .NET Framework to non-Windows platforms. Still, it's early in the game, and anything is possible. Whatever happens, having public standards and an open source implementation for their core technologies will certainly be a new experience for Microsoft.

A C# Example

Like most programming languages, C# defines data types, control structures, and more. Unlike older languages, however, C# does this by building on the CLR. Understanding the CLR therefore takes one a long way toward understanding C#. To illustrate this, here's a simple C# example:

```
// A C# example
interface IMath
{
       int Factorial(int f);
       double SquareRoot(double s);
}

class Compute : IMath
{
       public int Factorial(int f)
       {
              int i;
              int result = 1;
              for (i=2; i<=f; i++)
                 result = result * i;
       return result;
    }
```

```
    public double SquareRoot(double s)
    {
         return System.Math.Sqrt(s);
    }
}

class DisplayValues
{
    static void Main()
    {
         Compute c = new Compute();
         int v;
         v = 5;
         System.Console.WriteLine(
              "{0} factorial: {1}",
              v, c.Factorial(v));
         System.Console.WriteLine(
              "Square root of {0}: {1:f4}",
              v, c.SquareRoot(v));
    }
}
```

Every C# program is made up of one or more types

The program begins with a comment, indicated by two slashes, giving a brief description of the program's purpose. The body of the program consists of three types: an interface named IMath and the two classes Compute and DisplayValues. All C# programs consist of some number of types, the outermost of which must be classes, interfaces, structures, enums, or delegates. (Namespaces, discussed later, can also appear here.) All methods, fields, and other type members must belong to one of these types, which means that C# doesn't allow either global variables or global methods.

A C# interface is an expression of a CTS interface

The IMath interface, which is a C# incarnation of the Common Type System (CTS) interface type described in Chapter 3, defines the methods Factorial and SquareRoot. Each of these methods takes one parameter and returns a numeric result. These parameters are passed by value, the default in C#. This means that changes made to the parameter's value within the method won't be seen by the caller once the method returns. Placing the keyword ref in front of a parameter causes a parameter to be passed by reference, so any changes made within the method will be reflected back to the caller.

A C# class is an expression of a CTS class

Each class in this example is also a C# incarnation of the underlying CTS type. C# classes can implement one or more interfaces, inherit from at most one other class, and do all of the other things defined for a CTS class. The first class shown here, Compute, implements the IMath interface, as indicated by the colon between Compute and IMath. Accordingly, this class must contain implementations for both of the interface's methods. The body of the Factorial method declares a pair of integer variables, initializes the second of them to 1, then uses a simple for loop to calculate the factorial of its parameter (and doesn't bother to check for overflow, which is admittedly bad programming practice). Compute's second method, SquareRoot, is even simpler. It relies on the .NET Framework class library, calling the Sqrt function provided by the Math class in the System namespace.

Execution of a C# program begins with the method named Main

The last type in this simple example, the class DisplayValues, contains only a single method named Main. Much like C and C++, a C# program begins executing with this method in whatever type it appears. Although it's not shown here, Main can take arguments passed in when the program is started, and it must be declared as static. In this example, Main returns void, which is C#'s way of saying that the method has no return value. The type void cannot be used for parameters as in C and C++, however. Instead, its only purpose is to indicate that a method returns no value.

In this example, Main creates an instance of the Compute class using C#'s *new* operator. When this program is executed, new will be translated into the MSIL instruction newobj described in Chapter 3. Main next declares an int variable and sets its value to 5. This value is then passed as a parameter into calls to the Factorial and SquareRoot methods provided by the Compute instance. Factorial expects an int, which is exactly what's passed in this call, but SquareRoot expects a double. The int will automatically be converted into a double, since this conversion can be done with no loss of information. C# calls this

an *implicit* conversion, distinguishing it from type conversions that are marked explicitly in the code.

The Console class's WriteLine method writes formatted output to the console

The results are written out using the WriteLine method of the Console class, another standard part of the .NET Framework's System namespace. This method uses numbers that are wrapped in curly braces and that correspond to the variables to be output. Note that in the second call to WriteLine, the number in braces is followed by ":f4". This formatting directive means that the value should be written as a fixed-point number with four places to the right of the decimal. Accordingly, the output of this simple program is

```
5 factorial: 120
Square root of 5: 2.2361
```

The goal of this example is to give you a feeling for the general structure and style of C#. There's much more to the language, as the next sections illustrate.

C# Types

C# types are built on CTS types

Each type defined by C# is built on an analogous CTS type provided by the CLR. Table 4-1 shows most of the CTS types and their C# equivalents. As mentioned earlier in this book, all of these data types are defined in the System namespace. The C# equivalents shown here are in fact just shorthand synonyms for these alternative definitions. In the example just shown, for instance, the line

```
int i;
```

could have been replaced with

```
System.Int32 i;
```

Both work, and both produce exactly the same results.

Table 4-1 Some CTS Types and Their C# Equivalents

CTS	C#
Byte	byte
Char	char
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
UInt64	ulong
Single	float
Double	double
Decimal	decimal
Boolean	bool
Structure	struct
String	string
Class	class
Interface	interface
Delegate	delegate

Note that C# is case sensitive. Declaring a variable as "Double" rather than "double" will result in a compiler error. For people accustomed to languages derived from C, this will seem normal. To others, however, it might take a little getting used to.

Classes

Like a CTS class, a C# class can inherit directly from only one other class

C# classes expose the behaviors of a CTS class using a C-derived syntax. For example, CTS classes can implement one or more interfaces but inherit directly from at most one other class. A C# class Calculator that implements the interfaces IAlgebra

and ITrig and inherits from the class MathBasics would be declared as

```
class Calculator : MathBasics, IAlgebra, ITrig { ... }
```

Note that the class must come first in this list. C# classes can also be labeled as sealed or abstract, as defined in Chapter 3, and can be assigned public or internal visibility. These translate into the CTS-defined visibilities public and assembly, respectively. The default is internal. All of this information is stored in the metadata for this class once it has been compiled.

A C# class can include fields, methods, properties, constructors, destructors, and more

A C# class can contain fields, methods, and properties, all of which are defined for any CTS class. Each of these has an accessibility, which is indicated in C# by an appropriate access modifier such as public or private. It can also contain one or more constructors, called when an instance of this class is created, and at most one destructor, which is actually the name C# uses for a finalizer, a concept described in Chapter 3. If the class inherits from another class, it can potentially override one or more of the type members, such as a method, in its parent. To do this, the member being overridden must be declared as virtual.

C# supports operator overloading

A class can also define overloaded operators. An overloaded operator is one that has been redefined to have a special meaning when used with instances of this class. For example, a class representing workgroups in an organization might redefine the + operator to mean combining two workgroups into one.

Interfaces

A C# interface can inherit directly from one or more other interfaces

Interfaces are relatively simple things, and the basic C# syntax for describing an interface was shown in the earlier example. Not shown there was how C# expresses multiple interface inheritance, that is, one interface that inherits from more than one parent. If, for example, the interface ITrig inherits from the

three interfaces, ISine, ICosine, and ITangent, it could be declared as

```
Interface ITrig: ISine, ICosine, ITangent { ... }
```

ITrig will contain all the methods, properties, and other type members defined in its three parent interfaces as well as anything it defines on its own.

Structures

C# structures are like slightly simplified C# classes

Reflecting their definition in the CTS, structures in C# are much like classes. They can contain methods, fields, and properties and implement interfaces and more. They are value types rather than reference types, however, which means they're allocated on the stack. Value types also are prohibited from participating in inheritance. Unlike a class, a structure can't inherit from another type, and it's also not possible to define a type that inherits from a structure.

Here's a simple example of a C# structure:

```
struct employee
{
       string name;
       int age;
}
```

In this example, the structure contains only fields, much like a traditional C-style structure. Yet a structure can be much more complex. The Compute class shown earlier, for instance, could be converted to a structure, methods and all, by just changing the word *class* in its definition to *struct*. The program would function in just the same way.

Delegates

Passing a reference to a method as a parameter is often useful

Passing a reference to a method is a reasonably common thing to do. For example, suppose you need to tell some chunk of code what method in your code should be called when a spe-

cific event occurs. You need some way to pass in the identity of this callback function at runtime. In C and C++, you can do this by passing the address of the method, that is, a pointer to the code you want to be called. In the type-safe world of the .NET Framework, however, passing raw addresses isn't allowed. Yet the problem doesn't go away. A type-safe way to pass a reference to a method is still useful.

A C# delegate provides a type-safe way to pass a reference to a method

As described briefly in Chapter 3, the CTS defines the reference type *delegate* for this purpose. A delegate is an object that contains a reference to a method with a specific signature. Once it has been created and initialized, it can be passed as a parameter into some other method and then invoked. Here's a simple example of creating and using a delegate in C#:

```
delegate void SDelegate(string s);

class DelegateExample
{
    public static void Main()
    {
        SDelegate del = new SDelegate(WriteString);
        CallDelegate(del);
    }
    public static void CallDelegate(SDelegate Write)
    {
        System.Console.WriteLine("In CallDelegate");
        Write("A delegated hello");
    }
    public static void WriteString(string s)
    {
        System.Console.WriteLine("In WriteString:
             {0}", s);
    }
}
```

The example begins by defining SDelegate as a delegate type. This definition specifies that SDelegate objects can contain references only to methods that take a single string parameter. In the example's Main method, a variable `del` of type SDelegate is declared and then initialized to contain a reference to the WriteString method. This method is defined later in

the class, and as required, has a single parameter of type string. Main then invokes the CallDelegate method, passing in `del` as a parameter.

CallDelegate is defined to take an SDelegate as its parameter. In other words, what gets passed to this method is a delegate object that contains the address of some method. Because it's an SDelegate, that method must have a single parameter of type string. Inside CallDelegate, the method identified by the passed-in parameter is referred to as Write, and after printing a simple message, CallDelegate invokes this Write method. Because Write is actually a delegate, however, what really gets called is the method this delegate references, WriteString. The output of this simple example is

```
In CallDelegate
In WriteString: A delegated hello
```

Note that the CallDelegate method executes first, followed by WriteString.

A delegate can be combined with other delegates

Delegates can be significantly more complicated than this. They can be combined, for example, so that calling a single delegate results in calls to the two or more other delegates it contains. Yet even simple delegates can be useful. By providing a type-safe way to pass a reference to a method, they offer this important feature of C and C++ in a much less risky way.

Arrays

Like CTS arrays, C# arrays are reference types

As in other languages, C# arrays are ordered groups of elements of the same type. Unlike many other languages, however, C# arrays are objects. In fact, as described in Chapter 3, they are reference types, which means they get allocated on the heap. Here's an example that declares a single-dimensional array of integers:

```
int[] ages;
```

Since ages is an object, no instance exists until one is explicitly created. This can be done with

```
ages = new int[10];
```

which allocates space for ten integers on the heap. As this example shows, a C# array has no fixed size until an instance of that array type is created. It's also possible to both declare and create an array instance in a single statement, such as

```
int[] ages = new int[10];
```

Arrays of any type can be declared, but exactly how an array gets allocated depends on whether it's an array of value types or reference types. The example just shown allocates space for ten integers on the heap, while

```
string[] names = new string[10];
```

allocates space for ten references to strings on the heap. An array of value types, such as ints, actually contains the values, but an array of reference types, such as the strings in this example, contains only references to values.

C# arrays can be multidimensional

Arrays can also have multiple dimensions. For example, the statement

```
int[,] points = new int[10,20];
```

creates a two-dimensional array of integers. The first dimension has 10 elements, while the second has 20. Regardless of the number of dimensions in an array, however, the lower bound of each one is always zero.

Standard methods and properties can be accessed on all C# arrays

C#'s array type is built on the core array support provided by the CLR. Recall from the previous chapter that all CLR-based

arrays, including all C# arrays, inherit from System.Array. This base type provides various methods and properties that can be accessed on any instance of an array type. For example, the GetLength method can be used to determine the number of elements in a particular dimension of an array, while the CopyTo method can be used to copy all of the elements in a one-dimensional array to another one-dimensional array.

C# Control Structures

The control structures in C# are typical of a modern high-level language

C# provides the traditional set of control structures. Among the most commonly used of these is the if statement, which looks like this:

```
if (x > y)
   p = true;
else
   p = false;
```

Note that the condition for the if must be a value of type bool. It can't be an integer, as in C and C++.

C# also has a switch statement. Here's an example:

```
switch (x)
{
    case 1:
        y = 100;
        break;
    case 2:
        y = 200;
        break;
    default:
        y = 300;
        break;
}
```

Depending on the value of x, y will be set to 100, 200, or 300. The break statements cause control to jump to whatever statement follows this switch. Unlike C and C++, these (or similar) statements are mandatory in C#, even for the default case. Omitting them will produce a compiler error.

C# also includes various kinds of loops. In a while loop, the condition must evaluate to a bool rather than an integer value, which again is different from C and C++. There's also a do/while combination that puts the test at the bottom rather than at the top and a for loop, which was illustrated in the earlier example. Finally, C# includes a foreach statement, which allows iterating through all the elements in a value of a *collection* type. There are various ways a type can qualify as a collection type, the most straightforward of which is to implement the standard interface System.IEnumerable. A common example of a collection type is an array, and so one use of a foreach loop is to examine or manipulate each element in an array.

C# includes while, do/while, for, and foreach loops

C# also includes a goto statement, which jumps to a particular labeled point in the program, and a continue statement, which immediately returns to the top of whatever loop it's contained in and starts the next iteration. In general, the control structures in this new language are not very new, so they will be familiar to anybody who knows another high-level language.

Other C# Features

The fundamentals of a programming language are in its types and control structures. There are many more interesting things in C#, however—too many to cover in detail in this short survey. This section provides brief looks at some of the more interesting additional aspects of this new language.

Working with Namespaces

Because the underlying class libraries are so fundamental, namespaces are a critical part of programming with the .NET Framework. One way to invoke a method in the class libraries is by giving its fully qualified name. In the example shown earlier, for instance, the WriteLine method was invoked with

C#'s using statement makes it easier to reference the contents of a namespace

```
System.Console.WriteLine(...);
```

To lessen the amount of typing required, C# provides the *using* statement. This allows the contents of a namespace to be referenced with shorter names. It's common, for example, to start each C# program with the statement

```
using System;
```

If the example shown earlier had included this line, the WriteLine method could have been invoked with just

```
Console.WriteLine(...);
```

A program can also contain several using statements if necessary, as some of the examples later in this book will illustrate. It's also possible to define your own namespaces directly in C# containing types or even other namespaces. The types they contain can then also be referenced either with fully qualified names or through appropriate using statements.

Handling Exceptions

Exceptions provide a consistent way to handle errors across all CLR-based languages

Errors are a fact of life, at least for developers. In the .NET Framework, errors that occur at runtime are handled in a consistent way through exceptions. As in so much else, C# provides a syntax for working with exceptions, but the fundamental mechanisms are embedded in the CLR itself. This not only provides a consistent approach to error handling for all C# developers, but also means that all CLR-based languages will deal with this potentially tricky area in the same way. Errors can even be propagated across language boundaries as long as those languages are built on the .NET Framework.

An exception can be raised when an error occurs

An exception is an object that represents some unusual event, such as an error. The .NET Framework defines a large set of exceptions, and it's also possible to create custom exceptions. An exception is automatically raised by the runtime when errors occur. For example, in the code fragment

```
x = y/z;
```

what happens if z is zero? The answer is that the CLR raises the System.DivideByZeroException. If no exception handling is being used, the program will terminate.

Exceptions can be handled using try/catch blocks

C# makes it possible to catch exceptions, however, using try/catch blocks. The code above can be changed to look like this:

```
try
{
    x = y/z;
}
catch
{
    System.Console.WriteLine(“Exception caught”);
}
```

The code within the braces of the try statement will now be monitored for exceptions. If none occurs, execution will skip the catch statement and continue. If an exception occurs, however, the code in the catch statement will be executed, in this case printing out a warning, and execution will continue with whatever statement follows the catch.

Different exceptions can be handled differently

It’s also possible to have different catch statements for different exceptions and to learn exactly which exception occurred. Here’s another example:

```
try
{
    x = y/z;
}
catch (System.DivideByZeroException)
{
    System.Console.WriteLine("z is zero");
}
catch (System.Exception e)
{
    System.Console.WriteLine("Exception: {0}",
        e.Message);
}
```

In this case, if no exceptions occur, x will be assigned the value of y divided by z, and the code in both catch statements will be skipped. If z is zero, however, the first catch statement will be executed, printing a message to this effect. Execution will then skip the next catch statement and continue with whatever follows this try/catch block. If any other exception occurs, the second catch statement will be executed. This statement declares an object e of type System.Exception and then accesses this object's Message property to retrieve a printable string indicating what exception has occurred.

Custom exceptions can also be defined

Since CLR-based languages such as C# use exceptions consistently for error handling, why not define your own exceptions for handling your own errors? This can be done by defining a class that inherits from System.Exception and then using the throw statement to raise this custom exception. These exceptions can be caught with a try/catch block, just like those defined by the system.

Although it's not shown here, it's also possible to end a try/catch block with a finally statement. The code in this statement gets executed whether or not an exception occurs. This option is useful when some final cleanup must take place no matter what happens.

Using Attributes

A C# program can contain attributes

Once it's compiled, every C# type has associated metadata stored with it in the same file. Most of this metadata describes the type itself. As described in the previous chapter, however, metadata can also include attributes specified with this type. Given that the CLR provides a way to store attributes, it follows that C# must have some way to define attributes and their values. As described later in this book, attributes are used extensively by the .NET Framework class library. They can be applied to classes, interfaces, structures, methods, fields, para-

meters, and more. It's even possible to specify attributes that are applied to an entire assembly.

For example, suppose the Factorial method shown earlier had been declared with the WebMethod attribute applied to it. Assuming the appropriate using statements were in place to identify the correct namespace for this attribute, the declaration would look like this in C#:

```
[WebMethod] public int Factorial(int f) {...}
```

This attribute is used by ASP.NET, part of the .NET Framework class library, to indicate that a method should be exposed as a SOAP-callable Web service. (For more on how this attribute is used, see Chapter 7.) Similarly, including the attribute

```
[assembly:AssemblyCompanyAttribute("QwickBank")]
```

in a C# file will set the value of an assembly-wide attribute, one that gets stored in the assembly's manifest, containing the name of the company creating this assembly. This example also shows how attributes can have parameters, allowing their user to specify particular values for the attribute.

Custom attributes can also be defined

Developers can also create their own attributes. For example, you might wish to define an attribute that can be used to identify the date a particular C# type was modified. To do this, you can define a class that inherits from System.Attribute, then define the information you'd like that class to contain, such as a date. You can then apply this new attribute to types in your program and have the information it includes be automatically placed into the metadata for those types. Once they've been created, custom attributes can be read using the GetCustomAttributes method defined by the Attribute class, part of the System.Reflection namespace in the .NET Framework class library. Whether standard or custom, however, attributes are a commonly used feature in CLR-based software.

Writing Unsafe Code

C# developers typically rely on the CLR's garbage collection for memory management

C# normally relies on the CLR for memory management. When an instance of a reference type is no longer in use, for example, the CLR's garbage collector will eventually free the memory occupied by that type. As described in Chapter 3, the garbage collection process also rearranges the elements that are on the managed heap and currently in use, compacting them to free more space.

Pointers and garbage collection don't mix well

What would happen if traditional C/C++ pointers were used in this environment? A pointer contains a direct memory address, so a pointer into the managed heap would reference a specific location in the heap's memory. When the garbage collector rearranged the contents of the heap to create more free space, whatever the pointer pointed to could change. Blindly mixing pointers and garbage collection is a recipe for disaster.

C# allows creating unsafe code that uses pointers

Yet it's sometimes necessary. For example, suppose you need to call existing non-CLR-based code, such as the underlying operating system, and the call includes a structure with embedded pointers. Or perhaps a particular section of an application is so performance critical that you can't rely on the garbage collector to manage memory for you. For situations like these, C# provides the ability to use pointers in what's known as *unsafe code*.

Unsafe code can use pointers, with all of the attendant benefits and pitfalls pointers entail. To make this "unsafe" activity as safe as possible, however, C# requires that all code that does this be explicitly marked with the keyword unsafe. Within an unsafe method, the *fixed* statement can be used to lock one or more values of a reference type in place on the managed heap. (This is sometimes called *pinning* a value.) Here's a simple example:

```
class Risky
{
    unsafe public void PrintChars()
    {
```

```
        char[] charList = new char[2];
        charList[0] = 'A';
        charList[1] = 'B';

        System.Console.WriteLine("{0} {1}",
            charList[0], charList[1]);
        fixed (char* f = charList)
        {
            charList[0] = *(f+1);
        }
        System.Console.WriteLine("{0} {1}",
            charList[0], charList[1]);
    }
}

class DisplayValues
{
    static void Main()
    {
        Risky r = new Risky();
        r.PrintChars();
    }
}
```

The PrintChars method in the class Risky is marked with the keyword unsafe. This method declares the small character array charList and then sets the two elements in this array to "A" and "B," respectively. The first call to WriteLine produces

```
A  B
```

just as you'd expect. The fixed statement then declares a character pointer f and initializes it to contain the address of the charList array. Within the fixed statement's body, the first element of this array is assigned the value at address f+1. (The asterisk in front of the expression means "return what's at this address.") When WriteLine is called again, the output is

```
B  B
```

The value that is one beyond the start of the array, the character "B," has been assigned to the array's first position.

Unsafe code has limitations

This example does nothing useful, of course. Its intent is to make clear that C# does allow declaring pointers, performing pointer arithmetic, and more, as long as those statements are within areas clearly marked as unsafe. The language's creators really want you to be sure about doing this, so compiling any unsafe code requires specifying the /unsafe option to the C# compiler. Also, unsafe code can't be verified for type safety, which means that the CLR's built-in code access security features described in Chapter 3 can't be used. Unsafe code can be run in only a fully trusted environment, which makes it generally unsuitable for software that will be downloaded from the Internet. Still, there are cases when unsafe code is the right solution to a difficult problem.

Preprocessor Directives

Unlike C and C++, C# has no preprocessor. Instead, the compiler has built-in support for the most useful features of a preprocessor. For example, C#'s preprocessor directives include #define, a familiar term to C and C++ developers. This directive can't be used to define an arbitrary replacement string for a word, however—you can't define macros. Instead, #define is used to define only a symbol. That symbol can then be used together with the directive #if to provide conditional compilation. For example, in the code fragment

```
#define DEBUG
#if DEBUG
        // code compiled if DEBUG is defined
#else
        //code compiled if DEBUG is not defined
#endif
```

DEBUG is defined, so the C# compiler would process the code between the #if and #else directives. If DEBUG were undefined, something that's accomplished using the preprocessor directive #undef, the compiler would process the code between the #else and #endif directives.

Is C# Just a Copy of Java?

C# certainly does look a lot like Java. Given the additional similarities between the CLR and the Java virtual machine, it's hard to believe that Microsoft wasn't at least somewhat inspired by Java's success. By uniting C-style syntax with objects in a more approachable fashion than C++, Java's creators found the sweet spot for a large population of developers. I have seen projects that chose the Java environment rather than Microsoft technologies primarily because, unlike Java, neither Visual Basic 6 nor C++ was seen as a good language for large-scale enterprise development.

The arrival of C# and Visual Basic.NET will surely shore up Microsoft's technology against the Java camp. The quality of the programming language is no longer an issue. Yet this once again begs the question: Isn't C# like Java?

In many ways, the answer is yes. The core semantics of the CLR are very Java-esque. Being deeply object-oriented, providing direct support for interfaces, allowing multiple interface inheritance but only single implementation inheritance—these are all similar to Java. Yet C# also adds features that aren't available in Java. C#'s native support for properties, for instance, built on the support in the CLR, reflects the Visual Basic influence on C#'s creators. Attributes, also a CLR-based feature, provide a measure of flexibility beyond what Java offers, as does the ability to write unsafe code. Fundamentally, C# is an expression of the CLR's semantics in a C-derived syntax. Since those semantics are much like Java, C# is necessarily much like Java, too. But it's not the same language.

Is C# a better language than Java? There's no way to answer this question objectively, and it wouldn't matter if there were. Choosing a development platform based solely on the programming language is like buying a car because you like the radio. You can do it, but you'll be much happier if your decision takes into account the complete package.

If Sun had allowed Microsoft to modify Java a bit, my guess is that C# wouldn't exist today. For understandable reasons, however, Sun resisted Microsoft's attempts to customize Java for the Windows world. The result is two quite similar languages, each targeting a different development environment. Competition is good, and I'm confident that both languages will be in wide use five years from now.

C# is an attractive language. It combines a clean, concise design with a modern feature set. Although the world is littered with the carcasses of unsuccessful programming languages, C# isn't likely to join them. With Microsoft pushing it and its own quality pulling it, C# looks destined for a bright future.

Visual Basic.NET

Visual Basic is by a large margin the most popular programming language in the Windows world. Visual Basic.NET (VB.NET) brings enormous changes to this widely used tool. Like C#, VB.NET is built on the Common Language Runtime, and so large parts of the language are effectively defined by the CLR. In fact, except for their syntax, C# and VB.NET are largely the same language. Because both owe so much to the CLR and the .NET Framework class library, the functionality of the two is very similar.

Only Microsoft provides VB.NET compilers today

VB.NET can be compiled using Visual Studio.NET or vbc.exe, a command-line compiler supplied with the .NET Framework. Unlike C#, however, Microsoft has not submitted VB.NET to a standards body. Accordingly, while the open source world or some other third party could still create a clone, the Microsoft tools are likely to be the only viable choices for working in this language, at least for now.

A VB.NET Example

The quickest way to get a feeling for VB.NET is to see a simple example. The example that follows implements the same functionality as did the C# example shown earlier in this chapter. As you'll see, the differences from that example are largely cosmetic.

```
' A VB.NET example
Module DisplayValues
```

```
    Interface IMath
        Function Factorial(ByVal F As Integer) _
            As Integer
        Function SquareRoot(ByVal S As Double) _
            As Double
    End Interface

    Class Compute
        Implements IMath

        Function Factorial(ByVal F As Integer) _
            As Integer Implements IMath.Factorial
            Dim I As Integer
            Dim Result As Integer = 1

            For I = 2 To F
                Result = Result * I
            Next
            Return Result
        End Function

        Function SquareRoot(ByVal S As Double) _
            As Double Implements IMath.SquareRoot
            Return System.Math.Sqrt(S)
        End Function

    End Class

    Sub Main()
        Dim C As Compute = New Compute()
        Dim V As Integer

        V = 5
        System.Console.WriteLine( _
            "{0} factorial: {1}", _
            V, C.Factorial(V))
        System.Console.WriteLine( _
            "Square root of {0}: {1:f4}", _
            V, C.SquareRoot(V))
    End Sub

End Module
```

A Module provides a container for other VB.NET types

The example begins with a simple comment, indicated by the single quote that begins the line. Following the comment is an instance of the Module type that contains all of the code in this example. Module is a reference type, but it's not legal to create an instance of this type. Instead, its primary purpose is to provide a container for a group of VB.NET classes, interfaces, and

C# or VB.NET?

Before .NET, the language choice facing Microsoft-oriented developers was simple. If you were a hard-core developer, deeply proud of your technical knowledge, you embraced C++ in all its thorny glory. Alternatively, if you were more interested in getting the job done than in fancy technology and if that job wasn't too terribly complex or low level, you chose Visual Basic 6. Sure, the C++ guys abused you for your lack of linguistic savoir faire, but your code had a lot fewer obscure bugs.

This decade-old divide is over. C# and VB.NET are very nearly the same language. Except for relatively uncommon things such as writing unsafe code and operator overloading, they're equally powerful. Microsoft may change this in the future, making the feature sets of the two languages diverge. Until this happens, however (if it ever does), the main issue in making the choice is personal preference, which is really another way of saying "syntax."

Developers get very attached to how their language looks. C-oriented people love curly braces, while VB developers feel at home with Dim statements. Since many more developers use Visual Basic today than C++, I expect that VB.NET will be a more popular choice than C#. For the vast majority of VB developers who are fond of VB-style syntax, there's no reason to switch to C#. Even the .NET Framework documentation supplied by Microsoft is quite even-handed, usually providing examples in both languages. Given its much greater popularity today, I expect the dominant language for building Windows applications five years from now will still be Visual Basic.

In spite of this, however, I believe that any developer who knows C# can (and should) acquire at least a reading knowledge of VB.NET, and vice versa. The core semantics are identical, and after all, this is the really hard part of learning a language. In fact, to illustrate the near equality of these two languages, the examples in the following chapters of this book alternate more or less randomly between the two. In the world of .NET, you shouldn't think of yourself as a VB.NET developer or a C# developer. Whichever language you choose, you will in fact be a .NET Framework developer.

other types. In this case, the module contains an interface, a class, and a Sub Main procedure. It's also legal for a module to contain directly method definitions, variable declarations, and more that can be used throughout the module.

By default, VB.NET passes parameters by value, unlike Visual Basic 6

The module's interface is named IMath, and as in the earlier C# example, it defines the methods (or in the argot of Visual Basic, the functions) Factorial and SquareRoot. Each takes a single parameter, and each is defined to be passed by value, which means a copy of the parameter is made within the function. (The trailing underscore is the line continuation character, indicating that the following line should be treated as though no line break were present.) Passing by value is the default, so the example would work just the same without the ByVal indications. Passing by reference is the default in Visual Basic 6, which shows one example of how the language was changed to match the underlying semantics of the CLR.

A VB.NET class is an expression of a CTS class

The class Compute, which is the VB.NET expression of a CTS class, implements the IMath interface. Each of the functions in this class must explicitly identity the interface method it implements. Apart from this, the functions are just as in the earlier C# example except that a Visual Basic–style syntax is used. Note particularly that the call to System.Math.Sqrt is identical to its form in the C# example. C#, VB.NET, and any other language built on the CLR can access services in the .NET Framework class library in much the same way.

Execution begins in the Sub Main procedure

This simple example ends with a Sub Main procedure, which is analogous to C#'s Main method. The application begins executing here. In this example, Sub Main creates an instance of the Compute class using the VB.NET New operator (which will eventually be translated into the MSIL instruction newobj). It then declares an Integer variable and sets its value to 5.

As in the C# example, this simple program's results are written out using the WriteLine method of the Console class. Because

this method is part of the .NET Framework class library rather than any particular language, it looks exactly the same here as it did in the C# example. Not too surprisingly, then, the output of this simple program is

```
5 factorial: 120
Square root of 5: 2.2361
```

just as before.

VB.NET's similarities to Visual Basic 6 both help and hurt in learning this new language

To someone who knows Visual Basic 6, VB.NET will look familiar. To someone who knows C#, VB.NET will act in a broadly familiar way since it's built on the same foundation. But VB.NET is not the same as either Visual Basic 6 or C#. The similarities can be very helpful in learning this new language, but they can also be misleading. Be careful.

VB.NET Types

Like C#, the types defined by VB.NET are built on the CTS types provided by the CLR. Table 4-2 shows most of these types and their VB.NET equivalents.

VB.NET doesn't support all of the CTS types

Notice that some types, such as unsigned integers, are missing from VB.NET. Unsigned integers are a familiar concept to C++ developers but not to typical Visual Basic 6 developers. The core CTS types defined in the System namespace are available in VB.NET just as in C#, however, so a VB.NET developer is free to declare an unsigned integer using

```
Dim J As System.UInt32
```

Unlike C#, VB.NET is not case sensitive. There are some fairly strong conventions, however, which are illustrated in the example shown earlier. For people coming to .NET from Visual Basic 6, this case insensitivity will seem entirely normal. It's one example of why both VB.NET and C# exist, since the more a new

Table 4-2 Some CTS Types and Their VB.NET Equivalents

CTS	VB.NET
Byte	Byte
Char	Char
Int16	Short
Int32	Integer
Int64	Long
Single	Single
Double	Double
Decimal	Decimal
Boolean	Boolean
Structure	Structure
String	String
Class	Class
Interface	Interface
Delegate	Delegate

environment has in common with the old one, the more likely people will adopt it.

Classes

Like a CTS class, a VB.NET class can inherit directly from only one other class

VB.NET classes expose the behaviors of a CTS class using a VB-style syntax. Accordingly, VB.NET classes can implement one or more interfaces, but they can inherit from at most one other class. In VB.NET, a class Calculator that implements the interfaces IAlgebra and ITrig and inherits from the class MathBasics looks like this:

```
Class Calculator
    Inherits MathBasics
    Implements IAlgebra
    Implements ITrig
. . .
End Class
```

Note that, as in C#, the base class must precede the interfaces. Note also that any class this one inherits from might be written in VB.NET or in C# or perhaps in some other CLR-based language. As long as the language follows the rules laid down in the CLR's Common Language Specification, cross-language inheritance is straightforward. Also, if the class inherits from another class, it can potentially override one or more of the type members, such as a method, in its parent. This is allowed only if the member being overridden is declared with the keyword Overridable, analogous to C#'s keyword virtual.

VB.NET doesn't support operator overloading

VB.NET classes can be labeled as NotInheritable or MustInherit, which means the same thing as sealed and abstract, respectively, the terms used by the CTS and C#. VB.NET classes can also be assigned various accessibilities, such as Public and Friend, which largely map to visibilities defined by the CTS. A VB.NET class can contain variables, methods, properties, events, and more, just as defined by the CTS. Each of these can have an access modifier specified, such as Public, Private, or Friend. A class can also contain one or more constructors that get called whenever an instance of this class is created. Unlike C#, however, VB.NET does not support operator overloading. A class can't redefine what various standard operators mean when used with an instance of this class.

Interfaces

Like a CTS interface, a VB.NET interface can inherit directly from one or more other interfaces

Interfaces as defined by the CTS are a fairly simple concept. VB.NET essentially just provides a VB-derived syntax for expressing what the CTS specifies. Along with the interface behavior shown earlier, CTS interfaces can inherit from one or more other interfaces. In VB.NET, for example, defining an interface ITrig that inherits from the three interfaces, ISine, ICosine, and ITangent, would look like this:

```
Interface ITrig
    Inherits ISine
    Inherits ICosine
    Inherits ITangent
...
End Interface
```

Is Inheritance Really Worthwhile?

Inheritance is an essential part of object technology. Until .NET, Visual Basic didn't really support inheritance, and so (quite correctly) it was not viewed as an object-oriented language. VB.NET has inheritance, since it's built on the CLR, and so it is unquestionably truly object-oriented.

But is this a good thing? Microsoft certainly could have added inheritance to Visual Basic long ago, yet the language's keepers chose not to. Whenever I asked Microsoft why this was so, the answers revolved around two main points. First, inheritance can be tricky to understand and to get right. In a class hierarchy many levels deep, with some methods overridden and others overloaded, figuring out exactly what's going on isn't always easy. Given that the primary target audience for Visual Basic was not developers with formal backgrounds in computer science, it made sense to keep it simple.

The second point often made about why Visual Basic didn't have inheritance was that in many contexts, inheritance was not a good thing. This argument was made most strongly with COM, a technology that has no direct support for implementation inheritance. Inheritance binds a child class to its parent very closely, which means that a change in the parent can be catastrophic for the child. This "fragile base class" issue is especially problematic when the parent and child classes are written and maintained by completely separate organizations or when the parent's source isn't available to the creator of the child. In the component-oriented world of COM, this is a more than plausible argument.

So why has Microsoft apparently changed its mind about inheritance? Inheritance still can be problematic if changes in a parent class aren't communicated effectively to all developers who depend on that class, and it can also be complicated. The arguments Microsoft made are not incorrect. Yet the triumph of object technology is complete: Objects are everywhere. To create new languages in a completely new environment—that is, to create the .NET Framework—without full support for inheritance would brand any organization as irretrievably retro. And the benefits of inheritance, especially those gained by providing a large set of reusable classes such as the .NET Framework class library, are huge. The pendulum has swung, and inheritance is now essential.

Besides, most of the people in Redmond who argued against inheritance in the 1990s have probably retired by now. Never underestimate the power of new blood in a development group.

VB.NET structures can contain fields, provide methods, and more

Structures

Because both are based on the structure type defined by the CTS, structures in VB.NET are very much like structures in C#. Like a class, a structure can contain fields, members, and properties, implement interfaces, and more. VB.NET structures are value types, of course, which means that they can neither inherit from nor be inherited by another type. A simple employee structure might be defined in VB.NET as follows:

```
Structure Employee
     Public Name As String
     Public Age As Integer
End Structure
```

To keep the example simple, this structure contains only data members. As described earlier, however, CTS structures—and thus VB.NET structures—are in fact nearly as powerful as classes.

Delegates

The idea of passing an explicit reference to a procedure or function and then calling that procedure or function is not something that the typical Visual Basic programmer is accustomed to. Yet the CLR provides support for delegates, which allows exactly this. Why not make this support visible in VB.NET?

VB.NET allows creating and using delegates

VB.NET's creators chose to do this, allowing VB.NET programmers to create callbacks and other event-oriented code easily. Here's an example, the same one shown earlier in C#, of creating and using a delegate in VB.NET:

```
Module Module1

    Delegate Sub SDelegate(ByVal S As String)

    Sub CallDelegate(ByVal Write As SDelegate)
        System.Console.WriteLine("In CallDelegate")
        Write("A delegated hello")
    End Sub

    Sub WriteString(ByVal S As String)
        System.Console.WriteLine( _
            "In WriteString: {0}", S)
    End Sub

    Sub Main()
        Dim Del As New SDelegate( _
            AddressOf WriteString)
        CallDelegate(Del)
    End Sub

End Module
```

Although it's written in VB.NET, this code functions exactly like the C# example shown earlier in this chapter. Like that example, this one begins by defining SDelegate as a delegate type. As before, SDelegate objects can contain references only to methods that take a single String parameter. In the example's Sub Main method, a variable Del of type SDelegate is declared and then initialized to contain a reference to the WriteString subroutine. (A VB.NET subroutine is a method that, unlike a function, returns no result.) Doing this requires using VB.NET's AddressOf keyword before the subroutine's name. Sub Main then invokes CallDelegate, passing in Del as a parameter.

CallDelegate has an SDelegate parameter named Write. When Write is called, the method in the delegate that was passed into CallDelegate is actually invoked. In this example, that method is WriteString, so the code inside the WriteString procedure executes next. The output of this simple example is exactly the same as for the C# version shown earlier in this chapter:

```
In CallDelegate
In WriteString: A delegated hello
```

Is VB.NET Too Hard?

Maybe. There have been lots of complaints about the changes, and certainly some Visual Basic 6 developers will get left behind. Microsoft has historically targeted quite separate developer markets with Visual Basic and C++, yet with the .NET Framework, this distinction is greatly blurred. VB.NET and C# are functionally almost identical.

The .NET Framework is certainly simpler in many ways than the Windows DNA environment. The complexity of COM for cross-language calls is no longer required, for example. But the Framework is also harder for a certain class of developers, especially those with no formal training in computer science. One reason for Microsoft's success in the developer market was the approachability of Visual Basic. The people who create software tools often forget that they're almost always much better software developers than the people who will use those tools. As a result, they tend to create tools that they themselves would like to use, tools that are too complex for many of their potential customers.

The creators of Visual Basic never made this mistake. Despite the opprobrium heaped on the language and its users by C++ developers, Microsoft kept a clear focus on the developer population and skill level they wished to target. This was a good decision, as Visual Basic is now perhaps the world's most widely used programming language.

And yet many Visual Basic developers wanted more. VB.NET certainly gives them more, but it also requires *all* Visual Basic developers to step up a level in their technical knowledge. The skills required to build the GUI-based client of a two-tier application, the original target for this language, are almost entirely unrelated to what's needed to build today's scalable, multitier, Web-accessible solutions. Given this, perhaps the original audience Microsoft targeted for Visual Basic, some of the audience was just a step above power users, no longer has a role. With its complete object orientation and large set of more advanced features, VB.NET will certainly be too complex for many of them.

Yet building today's applications effectively was becoming more and more difficult with the old Visual Basic. Between a rock and a hard place, Microsoft chose to make this popular language both more powerful and more complex. Some developers will be very happy about this, but some won't. You can't please everybody, and the market will decide whether Microsoft has made the right decision.

Delegates are another example of the additional features Visual Basic has acquired from being rebuilt on the CLR. While this rethinking of the language certainly requires lots of learning from developers using it, the reward is a substantial set of features.

Arrays

Unlike Visual Basic 6, array indexes in VB.NET start at zero

Like arrays in C# and other CLR-based languages, arrays in VB.NET are reference types that inherit from the standard System.Array class. Accordingly, all of the methods and properties that class makes available are also usable with any VB.NET array. Arrays in VB.NET look much like arrays in earlier versions of Visual Basic. Perhaps the biggest difference is that the first member of a VB.NET array is referenced as element zero, while in previous versions of this language, the first member was element one. The number of elements in an array is thus one greater than the number that appears in its declaration. For example, the following statement declares an array of eleven integers:

```
Dim Ages(10) As Integer
```

Unlike C#, there's no need to create explicitly an instance of the array using New. It's also possible to declare an array with no explicit size and later use the ReDim statement to specify how big it will be. For example, this code

```
Dim Ages() As Integer
ReDim Ages(10)
```

results in an array of eleven integers just as in the previous example. Note that the index for both of these arrays goes from 0 to 10, not 1 to 10.

VB.NET also allows multidimensional arrays. For example, the statement

```
Dim Points(10,20) As Integer
```

creates a two-dimensional array of integers with 11 and 21 elements, respectively. Once again, both dimensions are zero-based, which means that the indexes go from 0 to 10 in the array's first dimension and 0 to 20 in the second dimension.

VB.NET Control Structures

VB.NET's control structures will look familiar to most developers

While the CLR says a lot about what a .NET Framework–based language's types should look like, it says essentially nothing about how that language's control structures should look. Accordingly, adapting Visual Basic to the CLR required making changes to VB's types, but the language's control structures are fairly standard. An If statement, for example, looks like this:

```
If (X > Y) Then
    P = True
Else
    P = False
End If
```

while a Select Case statement analogous to the C# switch shown earlier looks like this:

```
Select Case X
    Case 1
        Y = 100
    Case 2
        Y = 200
    Case Else
        Y = 300
End Select
```

As in the C# example, different values of x will cause y to be set to 100, 200, or 300. Although it's not shown here, the Case clauses can also specify a range rather than a single value.

VB.NET includes a While loop, a Do loop, a For...Next loop, and a For Each loop

The loop statements available in VB.NET include a While loop, which ends when a specified Boolean condition is no longer true; a Do loop, which allows looping until a condition is no longer true or until some condition becomes true; and a For...Next loop, which was shown in the example earlier in this

section. And like C#, VB.NET includes a For Each statement, which allows iterating through all the elements in a value of a collection type.

VB.NET also includes a goto statement, which jumps to a labeled point in the program, and a few more choices. The innovation in the .NET Framework doesn't focus on language control structures (in fact, it's not easy to think of the last innovation in language control structures), and so VB.NET doesn't offer much that's new in this area.

Other VB.NET Features

VB.NET exposes most of the CLR's features

The CLR provides many other features, as seen in the description of C# earlier in this chapter. With very few exceptions, the creators of VB.NET chose to provide these features to developers working in this newest incarnation of Visual Basic. This section looks at how VB.NET provides some more advanced features.

Working with Namespaces

VB.NET's Imports statement makes it easier to reference the contents of a namespace

As mentioned in Chapter 3, namespaces aren't directly visible to the CLR. Just as in C#, however, they are an important part of writing applications in VB.NET. As shown earlier in the VB.NET example, access to classes in .NET Framework class library namespaces looks just the same in VB.NET as in C#. Because the Common Type System is used throughout, methods, parameters, return values, and more are all defined in a common way. Yet how a VB.NET program indicates which namespaces it will use is somewhat different from how it's done in C#. Commonly used namespaces can be identified for a module with the Imports statement. For example, preceding a module with

```
Imports System
```

would allow invoking the System.Console.WriteLine method with just

```
Console.WriteLine( . . .)
```

VB.NET's Imports statement is analogous to C#'s using statement. Both allow developers to do less typing. And as in C#, VB.NET also allows defining and using custom namespaces.

Handling Exceptions

One of the greatest benefits of the CLR is that it provides a common way to handle exceptions across all .NET Framework languages. This common approach allows errors to be found in, say, a C# routine and then is handled in code written in VB.NET. The syntax for how these two languages work with exceptions is different, but the underlying behavior, specified by the CLR, is the same.

As in C#, try/catch blocks are used to handle exceptions in VB.NET

Like C#, VB.NET uses Try and Catch to provide exception handling. Here's a VB.NET example of handling the exception raised when a division by zero is attempted:

```
Try
     X = Y/Z
Catch
     System.Console.WriteLine("Exception caught")
End Try
```

Any code between the Try and Catch is monitored for exceptions. If no exception occurs, execution skips the Catch clause and continues with whatever follows End Try. If an exception occurs, the code in the Catch clause is executed, and execution continues with what follows End Try.

VB.NET offers essentially the same exception handling options as C#

As in C#, different Catch clauses can be created to handle different exceptions. A Catch clause can also contain a When clause with a Boolean condition. In this case, the exception will be caught only if that condition is true. Also like C#, VB.NET allows defining your own exceptions and then raising them with the Throw statement. VB.NET also has a Finally statement. As in C#, the code in a Finally block is executed whether or not an exception occurs.

Using Attributes

A VB.NET program can contain attributes

Code written in VB.NET is compiled into MSIL, so it must have metadata. Because it has metadata, it also has attributes. The designers of the language provided a VB-style syntax for specifying attributes, but the end result is the same as for any CLR-based language: Extra information is placed in the metadata of some assembly. To repeat once again an example from earlier in this chapter, suppose the Factorial method shown in the complete VB.NET example had been declared with the WebMethod attribute applied to it. This attribute instructs the .NET Framework to expose this method as a SOAP-callable Web service, as described in more detail in Chapter 7. Assuming the appropriate Imports statements were in place to identify the correct namespace for this attribute, the declaration would look like this in VB.NET:

```
<WebMethod()> Public Function Factorial(ByVal F _
As Integer) As Integer Implements IMath.Factorial
```

This attribute is used by ASP.NET to indicate that a method contained in an .asmx page should be exposed as a SOAP-callable Web service. Similarly, including the attribute

```
<assembly:AssemblyCompanyAttribute("QwickBank")>
```

in a VB.NET file will set the value of an attribute stored in this assembly's manifest that identifies QwickBank as the company that created this assembly. VB.NET developers can also create their own attributes by defining classes that inherit from System.Attribute and then have whatever information is defined for those attributes automatically copied into metadata. As in C# or another CLR-based language, custom attributes can be read using the GetCustomAttributes method defined by the System.Reflection namespace's Attribute class.

VB.NET and C# offer very similar features

Attributes are just one more example of the tremendous semantic similarity of VB.NET and C#. While they look quite

different, the capabilities of the two languages are very similar. Which one a developer prefers will be largely an aesthetic decision.

Why Provide All of These Languages?

Microsoft says that more than twenty languages have been ported to the CLR. Along with the languages shipped by Microsoft itself, programmers will have plenty of options to choose from. Yet given the CLR's central role in defining these languages, they often have much in common. What's the real benefit of having multiple languages based on the CLR?

There are two key advantages. First, the existing pre-.NET population of Windows developers is split into two primary language camps: C++ and Visual Basic. Microsoft needs to move both groups of developers forward, and both certainly have some attachment to their language. Although the semantics of the CLR (and of languages built on it such as C# and Visual Basic.NET) are different from either C++ or Visual Basic 6, the fundamental look of these new languages will be familiar. If Microsoft chose to provide only, say, C#, it's a safe bet that developers who were wedded to Visual Basic 6 would probably be resistant to moving to .NET. Similarly, providing only a CLR-based language derived from Visual Basic wouldn't make C++ developers very happy. People who write code get attached to the oddest things (curly braces, for example), and so providing both C# and Visual Basic.NET is a good way to help the current Windows developer population move forward.

The second benefit in providing multiple languages is that it gives the .NET Framework something the competition doesn't have. One complaint about the Java world has been that it requires all developers always to use the same language. The .NET Framework's multilingual nature offers more choice, so it gives Microsoft something to tout over its competitors.

In fact, however, there are some real benefits to having just one language. Why add extra complexity, such as a different syntax for expressing the same behavior, when there's no clear benefit? Java's one-language-all-the-time approach has the virtue of simplicity. Even in the .NET world, organizations

would do well to avoid multilanguage projects if possible. This isn't the problem it was with Windows DNA, since code written in different CLR-based languages can interoperate with no problems. Developers who know C# should also have no trouble understanding VB.NET, and vice versa. Still, having two or more separate development groups using distinct languages will complicate both the initial project and the maintenance effort that follows. It's worth avoiding if possible.

In the end, the diverse set of languages announced for the .NET Framework probably won't matter much. Because of Microsoft's strong support, expressed most powerfully in Visual Studio.NET, C# and Visual Basic.NET will be dominant for creating new CLR-based applications. The other languages might be interesting for universities, but for professional developers, good tools are essential. Most Windows developers today believe that Visual Studio is the best tool for building code on Windows. Just as in the pre-.NET world, I expect Visual Studio.NET and the languages it supports to be the dominant choices for Windows developers.

C++ with Managed Extensions

C++ was too popular for the .NET Framework's creators to ignore

C++ is a very popular language, one that's been in wide use for more than a dozen years. Providing some way to use C++ with the .NET Framework is essential. Yet the semantics of C++ don't exactly match those of the CLR. They have much in common—both are object-oriented, for example—but there are also many differences. C++, for instance, supports multiple inheritance, the ability of a class to inherit simultaneously from two or more parent classes, while the CLR does not.

Unlike Visual Basic, Microsoft isn't free to change C++ unilaterally to fit the CLR

Visual Basic 6 also differs substantially from the CLR, but Microsoft owns Visual Basic. The company was free to change it as they wished, so VB.NET was designed to match the CLR. Microsoft does not own C++, however. Unilaterally changing the language to match the CLR would have met with howls of protest. Yet providing no way to create .NET Framework–based applications in C++ would also have left many developers very unhappy. What's the solution?

Microsoft has defined a set of Managed Extensions for C++

The answer Microsoft chose was to create a set of extensions to the base C++ language. Officially known as *Managed Extensions for C++*, the resulting dialect is commonly referred to as just *Managed C++*. C++ is not simple to begin with, and Managed C++ adds new complexities. The goal of this chapter's final section is to provide an overview of the changes wrought by Managed C++.

Managed C++ defines several new keywords

Before looking at a Managed C++ example, it's useful to describe some of the extensions made to the language. In particular, several keywords have been added to allow access to CLR services, all of which begin with two underscores. (This follows

Managed C++ or C#?

C++ has legions of die-hard fans. And why shouldn't it? It's a powerful, flexible tool for building all kinds of applications. It's complicated, too, which means that learning to exploit all that power and flexibility takes a substantial amount of effort. Anyone who's put in the time to master C++ is bound to be less than thrilled about leaving it behind.

Yet for brand-new applications built from scratch on the .NET Framework, C++ probably should be left behind. For a C++ developer, learning C# isn't difficult. In fact, learning C# will probably be easier than using Managed C++ to write .NET Framework–based applications. As the short summary in this chapter suggests, Managed C++ adds even more complexity to an already complex language. For new applications, C# is probably a better choice.

For extending existing C++ applications with managed code, however, Managed C++ is a good choice. And if you plan to port an existing C++ application to run on the Framework, Managed C++ is also a good choice, since it saves you from rewriting large parts of your code. Although it's not as important in the .NET Framework world as either VB.NET or C#, Managed C++ is nevertheless a significant member of .NET's language arsenal.

the convention defined in the ANSI standard for C++ extensions.) Among the most important of these are the following:

- **__gc:** Indicates that a type is subject to garbage collection. In other words, this keyword means that the type being declared is a CTS reference type. Managed C++ allows this keyword to be applied to classes, arrays, and other types.
- **__value:** Indicates that a type is not subject to garbage collection; that is, that the type is a CTS value type.
- **__interface:** Used to define a CTS interface type.
- **__box:** An operation that converts a CTS value type to a reference type.
- **__unbox:** An operation that converts a boxed CTS value type back to its original form.
- **__delegate:** Used to define a CTS delegate type.

Given this brief introduction, we can now make some sense out of an example.

A Managed C++ Example

C# and VB.NET were both designed for the CLR, while C++ was not. As a result, code written in Managed C++ can look a bit odd. Here's the same example shown earlier in this chapter, this time in Managed C++:

```
// A Managed C++ example
#include "stdafx.h"
#using <mscorlib.dll>

__gc __interface IMath
{
       int Factorial(int f);
       double SquareRoot(double s);
};
```

```
__gc class Compute : public IMath
{
    public: int Factorial(int f)
    {
        int i;
        int result = 1;
        for (i=2; i<=f; i++)
              result = result * i;
        return result;
    };

    public: double SquareRoot(double s)
    {
        return System::Math::Sqrt(s);
    }
};

void main(void)
{
    Compute *c = new Compute;
    int v;
    v = 5;
  System::Console::WriteLine(
       "{0} factorial: {1}",
       box(v), __box(c->Factorial(v)));
  System::Console::WriteLine(
       "Square root of {0}: {1:f4}",
       box(v), __box(c->SquareRoot(v)));
}
```

Managed C++ resembles C#

The first thing to notice is how much this example resembles the C# version. Most of the basic syntax and many of the operators are the same. Yet it's different, too, beginning with the #include and #using statements necessary for creating managed code in C++. Following these, the interface IMath is defined, just as before. This time, however, it uses the __interface keyword and precedes it with the __gc keyword. The result is a C++ incarnation of a CTS-defined interface.

Next comes the class Compute, which implements the IMath interface. This class too is declared with the __gc keyword, which means that it's a CTS class with a lifetime managed by the CLR rather than the developer. The class varies a bit in syntax from the C# example, since C++ doesn't express things in exactly the same way, but it's nonetheless very similar.

Managed C++ requires explicit boxing

The example ends with a standard C++ main function. Just as before, it creates an instance of the Compute class and then calls its two methods, all using standard C++ syntax. The only substantive difference is in the calls to WriteLine. Because this method expects reference parameters, the __box operator must be used to pass the numeric parameters correctly. Boxing also occurred for this parameter in C# and VB.NET, but it was done automatically. Because C++ was not originally built for the CLR, however, the developer must explicitly request this operation. Finally, just as you'd expect, the output of this example is the same as before: the factorial and square root of five.

Managed C++ Types

Managed and unmanaged C++ code can coexist in a process

Managed C++ allows full access to the .NET Framework, including the types defined by the CLR and more. It's important to note that managed and unmanaged code, classes defined with and without __gc, can be defined in the same file, and they can exist in the same running process. Only the managed classes are subject to garbage collection, however; unmanaged classes must be explicitly freed as usual in C++. Table 4-3 shows some of the major CLR types and their equivalents in Managed C++.

Other Managed C++ Features

Managed C++ allows full access to the CLR's features

Because it fully supports the CLR, there's much more in Managed C++. Delegates can be created using the *__delegate* keyword, while namespaces can be referenced with a *using namespace* statement, such as

```
using namespace System;
```

Exceptions can be handled using try/catch blocks, and custom CLR exceptions that inherit from System::Exception can be created. Attributes can also be embedded in code using the same syntax as in C#.

Table 4-3 Some CLR Types and Their Managed C++ Equivalents

CLR	Managed C++
Byte	unsigned char
Char	wchar_t
Int16	short
Int32	int, long
Int64	__int64
UInt16	unsigned short
UInt32	unsigned int, unsigned long
UInt64	unsigned __int64
Single	float
Double	double
Decimal	Decimal
Boolean	bool
Structure	struct
String	String*
Class	__gc class
Interface	__gc __interface
Delegate	__delegate

C++ is the only language in Visual Studio that can compile directly to native code

Managed C++ is a major extension to the C++ environment provided by Visual Studio.NET, but it's not the only new feature. This latest edition of Microsoft's flagship development tool also includes better support for building traditional applications, including COM-based applications. Except for C++, all languages in Visual Studio.NET compile only to MSIL, and they require the .NET Framework to run. Since all Managed C++ classes are compiled to MSIL, the language can obviously be used to generate Framework-based code, but C++ is unique in that it also allows compiling directly to a machine-specific

Is C++ a Dead Language?

C++ has been the workhorse of professional software developers for most of the last decade. It's been used to write Lotus Notes, a surfeit of business applications, and even parts of Windows. Yet in a world that offers C#, VB.NET, and Java, where does C++ fit? Has its usefulness come to an end?

Certainly not. C#, VB.NET, and Java are much better than C++ for many types of applications, even many for which C++ has commonly been used. But all three of these languages operate in a virtual machine environment. This has many benefits, but there's also a substantial price: performance and size. Some categories of applications, especially system-level software, can't afford this. Who's going to build an operating system in a garbage-collected language? Who wants to build embedded applications for memory-constrained devices in a language that requires a large supporting runtime library?

The day when C++ was the default choice for building a broad range of new applications is over. In the Microsoft world, C# and VB.NET will be the new defaults, while Java dominates elsewhere. Yet in cases where none of these is appropriate—and they do exist—C++ will still dominate. Its role will surely shrink, probably substantially, but C++ is not about to disappear.

binary. For building applications that don't require the CLR, C++ is the only way to go.

Conclusion

The .NET Framework brings a new approach to programming language design

Programming languages are a fascinating topic. There appears to be wide agreement on what fundamental features a modern programming language should have and how it should behave. These features and behaviors are essentially what the CLR provides. There is little agreement on how a modern programming language should look, however, with everyone voting for his or her preferred syntax. By providing a common implementation

of the core and then allowing diverse expressions of that core, the .NET Framework brings a new approach to language design. Even without Microsoft's backing, this would be an attractive model for creating a development environment. With the backing of the world's largest software company, it's bound to affect the lives of many, many developers.